Balanced Trees

- 2-3-4 trees
- red-black trees
- B-trees

References:
Algorithms in Java, Chapter 13
`http://www.cs.princeton.edu/algs4/43balanced`

Symbol table review

Symbol table. Key-value pair abstraction.
- Insert a value with specified key.
- Search for value given key.
- Delete value with given key.

Randomized BST.
- Probabilistic guarantee of ~ c lg N time per operation.
- Need subtree count in each node.
- Need random numbers for each insertion and deletion.

This lecture. 2-3-4 trees, left-leaning red-black trees, B-trees.

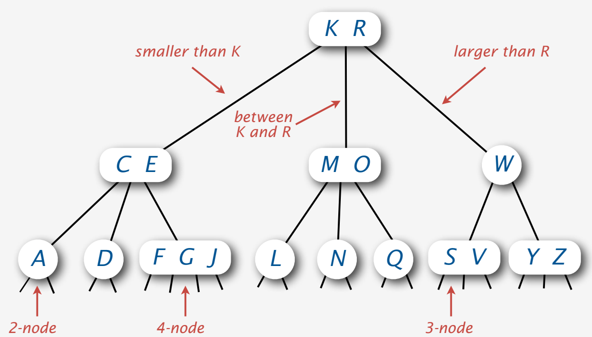
introduced to the world in
COS 226, Fall 2007
(sorry, no handouts currently available)

- 2-3-4 trees
- red-black trees
- B-trees

2-3-4 tree

Allow 1, 2, or 3 keys per node.
- 2-node: one key, two children.
- 3-node: two keys, three children.
- 4-node: three keys, four children.

Maintain perfect balance. Every path from root to leaf has same length.

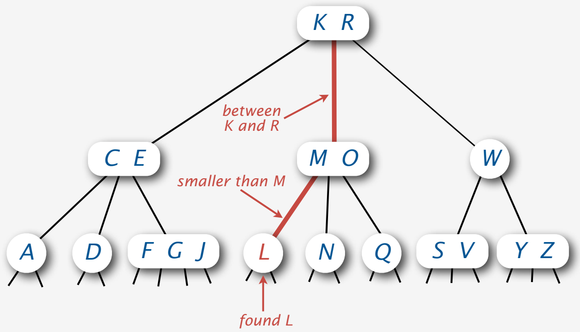


Search in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

Ex. Search for L.

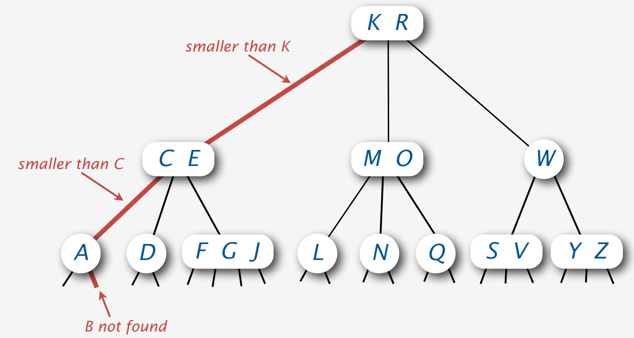


Insertion in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key.
- Follow associated link (recursively).

Ex. Search for B.

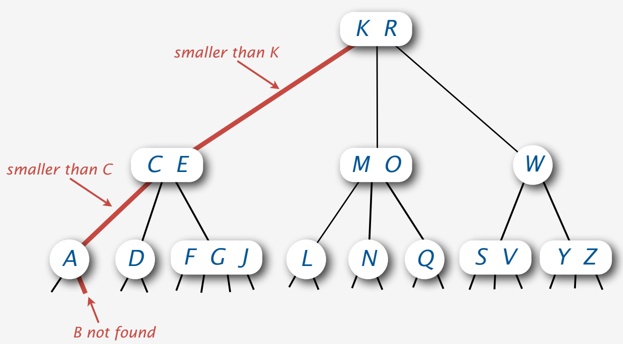


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.

Ex. Insert B.

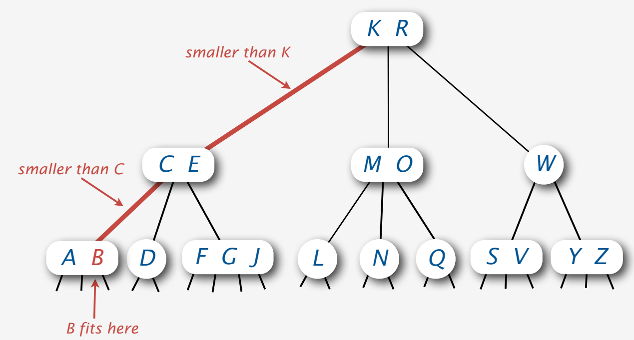


Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node.

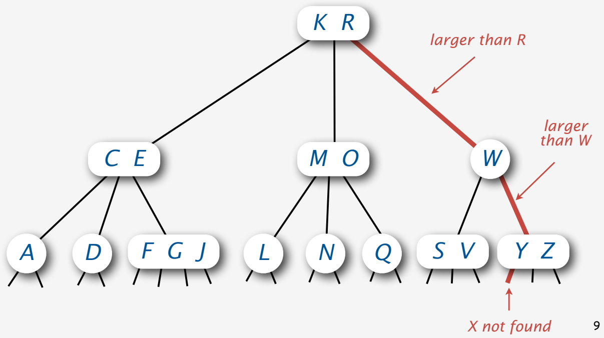
Ex. Insert B.



Insertion in a 2-3-4 tree

Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.

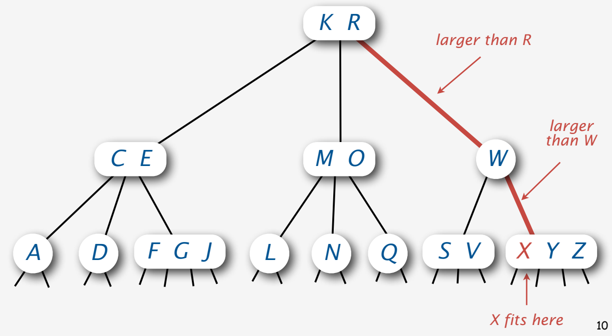
Ex. Insert X.



Insertion in a 2-3-4 tree

Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.

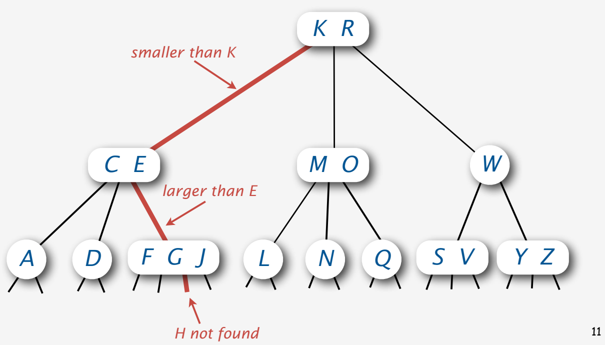
Ex. Insert X.



Insertion in a 2-3-4 tree

Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.

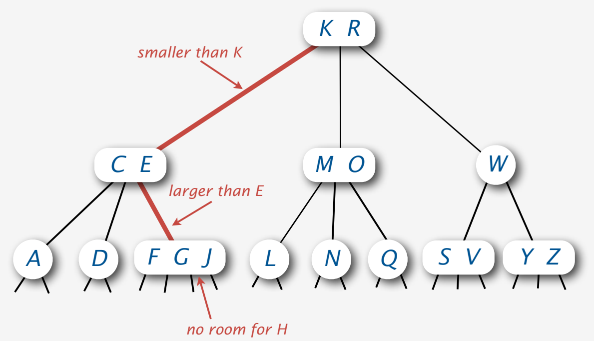
Ex. Insert H.



Insertion in a 2-3-4 tree

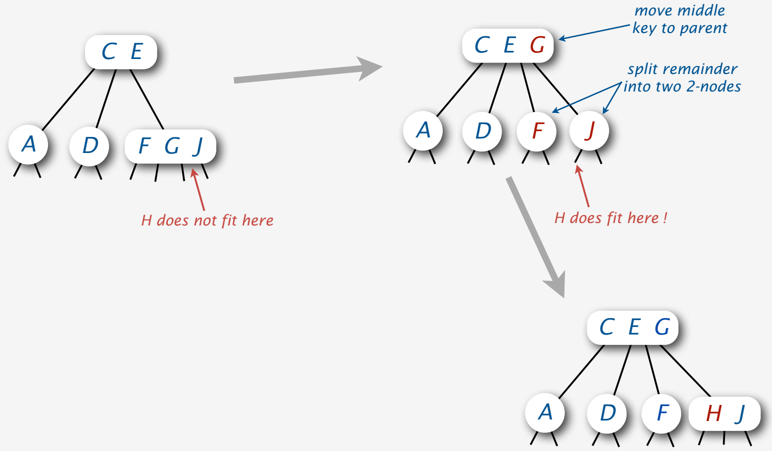
Insert.
- Search to bottom for key.
- 2-node at bottom: convert to 3-node.
- 3-node at bottom: convert to 4-node.
- 4-node at bottom: no room for new key!

Ex. Insert H.



Splitting a 4-node in a 2-3-4 tree

Idea. Split the 4-node to make room.

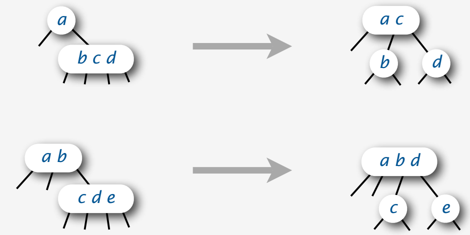


Problem. Doesn't work if parent is a 4-node.

Splitting 4-nodes in a 2-3-4 tree

Strategy. Split 4-nodes on the way down the tree.

Invariant. Current node is not a 4-node.



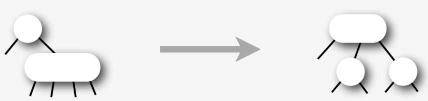
transformations to split a 4-node

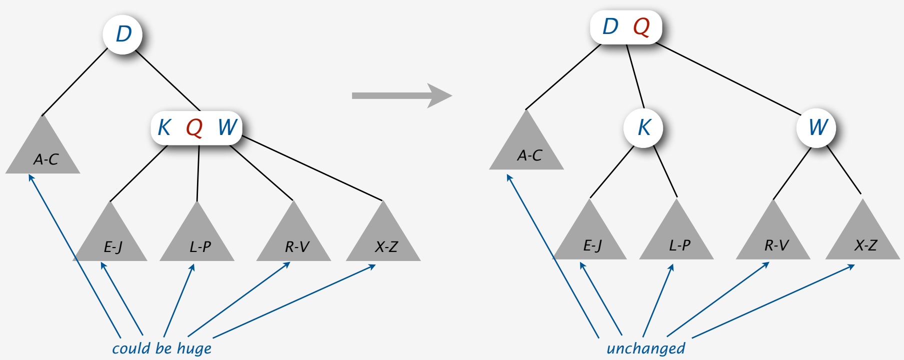
Consequences.

- 4-node below a 4-node case never happens.
- Insertion at bottom node is easy since it's not a 4-node.

Splitting a 4-node below a 2-node in a 2-3-4 tree

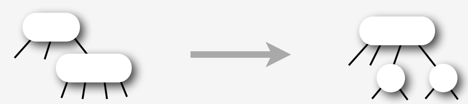
A local transformation that works anywhere in the tree.

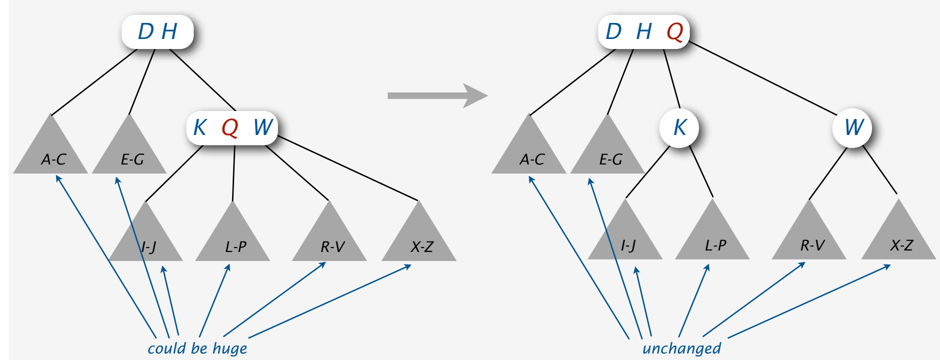




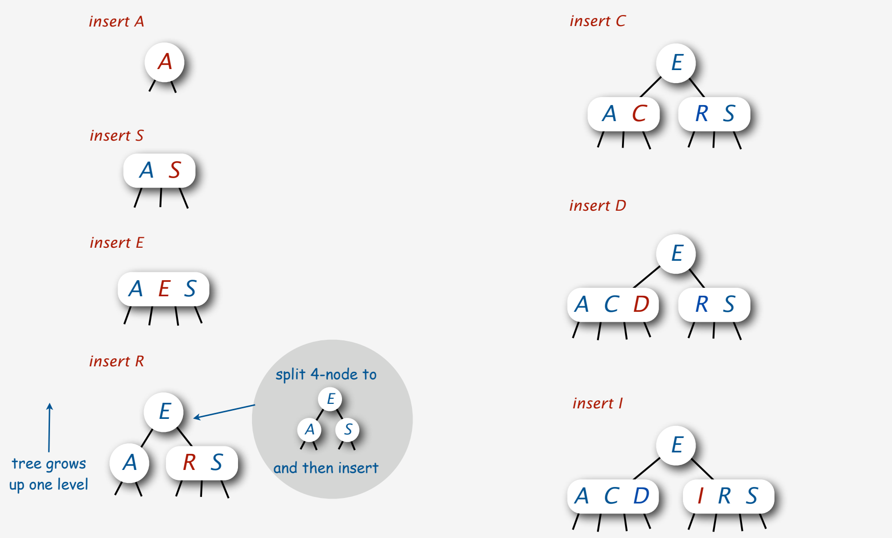
Splitting a 4-node below a 3-node in a 2-3-4 tree

A local transformation that works anywhere in the tree.

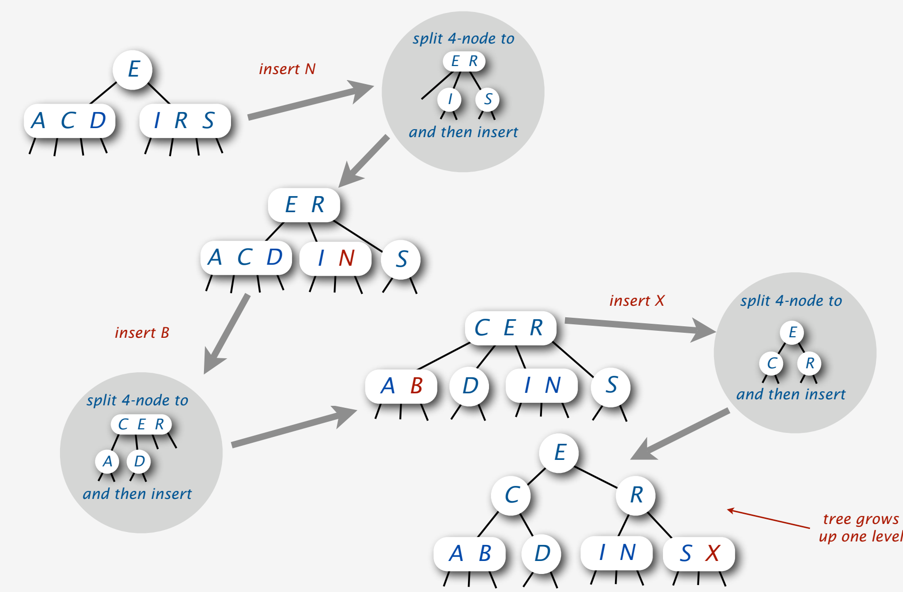




Growth of a 2-3-4 tree

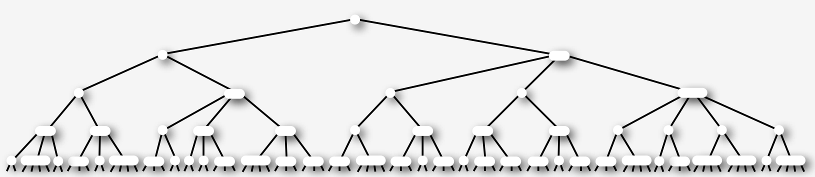


Growth of a 2-3-4 tree (cont)



Balance in a 2-3-4 tree

Key property. All paths from root to leaf have same length.

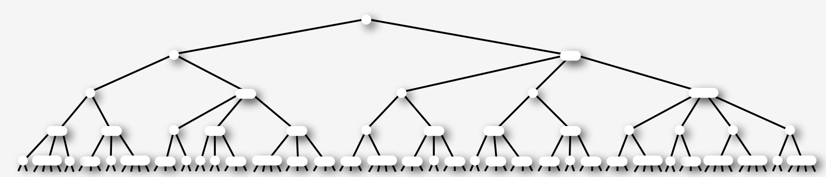


Tree height.

- Worst case:
- Best case:

Balance in a 2-3-4 tree

Key property. All paths from root to leaf have same length.



Tree height.

- Worst case: lg N. [all 2-nodes]
- Best case: $\log_4 N = 1/2 \lg N$. [all 4-nodes]
- Between 10 and 20 for a million nodes.
- Between 15 and 30 for a billion nodes.

Guaranteed logarithmic performance for search and insert.

2-3-4 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Large number of cases for splitting.

```
private void insert(Key key, Value val)
{
   if (root.is4Node()) root.split4Node();
   Node x = root;
   while (x.getChild(key) != null)
   {
      x = x.getChild(key))
      if (x.is4Node()) x.split4Node();
   }
   if      (x.is2Node()) x.make3Node(key, val);
   else if (x.is3Node()) x.make4Node(key, val);
}
```

fantasy code

Bottom line. Could do it, but there's a better way.

ST implementations: summary

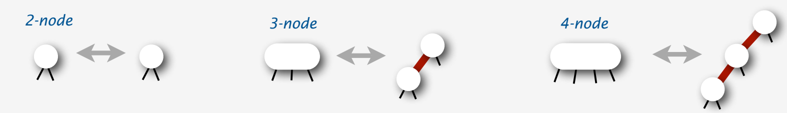
implementation	guarantee		average case		ordered iteration?	operations on keys
	search	insert	search hit	insert		
unordered array	N	N	N/2	N	no	`equals()`
unordered list	N	N	N/2	N	no	`equals()`
ordered array	lg N	N	lg N	N/2	yes	`compareTo()`
ordered list	N	N	N/2	N/2	yes	`compareTo()`
BST	N	N	1.38 lg N	1.38 lg N	yes	`compareTo()`
randomized BST	3 lg N	3 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
2-3-4 tree	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`

constants depend upon implementation

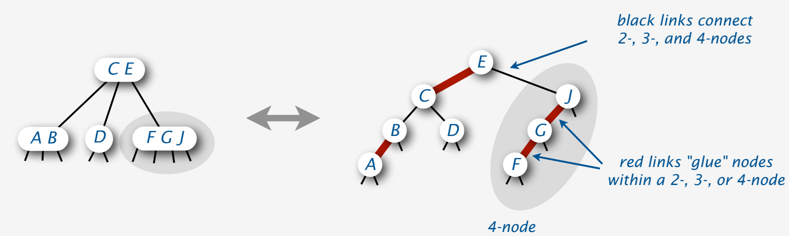
- 2-3-4 trees
- **red-black trees**
- B-trees

Left-leaning red-black trees (Guibas-Sedgewick, 1979 and Sedgewick, 2007)

1. Represent 2-3-4 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3- and 4-nodes.

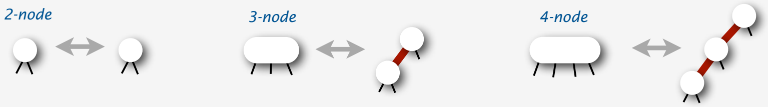


Key property. 1-1 correspondence between 2-3-4 and LLRB.



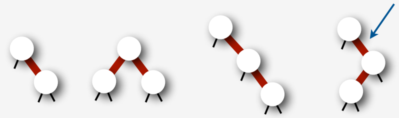
Left-leaning red-black trees (Guibas-Sedgewick, 1979 and Sedgewick, 2007)

1. Represent 2-3-4 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3- and 4-nodes.

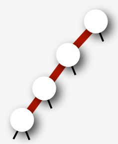


Disallowed.

- Right-leaning red link.



- Three red links in a row.

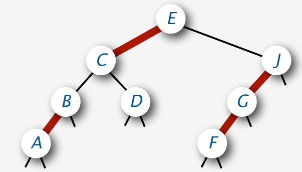


Search implementation for red-black trees

Observation. Search is the same as for elementary BST (ignore color).

but runs faster because of better balance

```
public Val get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if       (cmp  < 0) x = x.left;
      else if  (cmp  > 0) x = x.right;
      else if  (cmp == 0) return x.val;
   }
   return null;
}
```



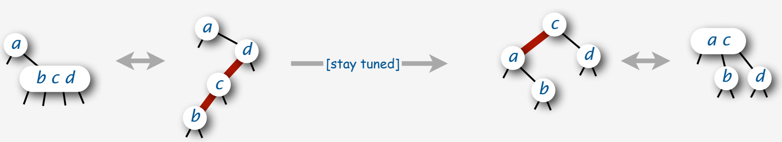
Remark. Many other ops (e.g., iteration, kth largest, ceiling) are also the same.

Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3-4 trees.

- Inserting a node at the bottom.

- Splitting a 4-node.

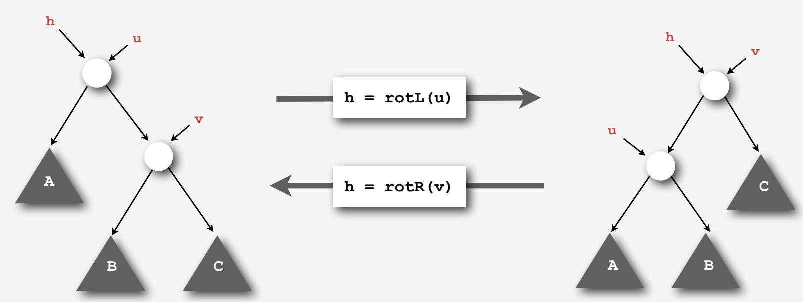


Potential concern. Lots of cases to consider.

Review: rotation in a BST

Two fundamental operations to rearrange nodes in a BST.

- Maintain symmetric order.
- Local transformations (change just 3 pointers).



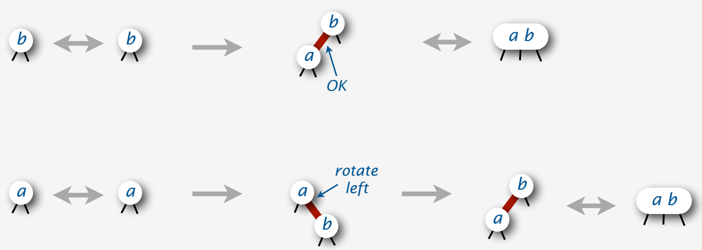

```
private Node rotL(Node h)
{  /* as before */  }
```

```
private Node rotR(Node h)
{  /* as before */  }
```

Insertion in a LLRB tree: adding a node to the bottom

Invariant. Node at bottom is a 2-node or a 3-node.

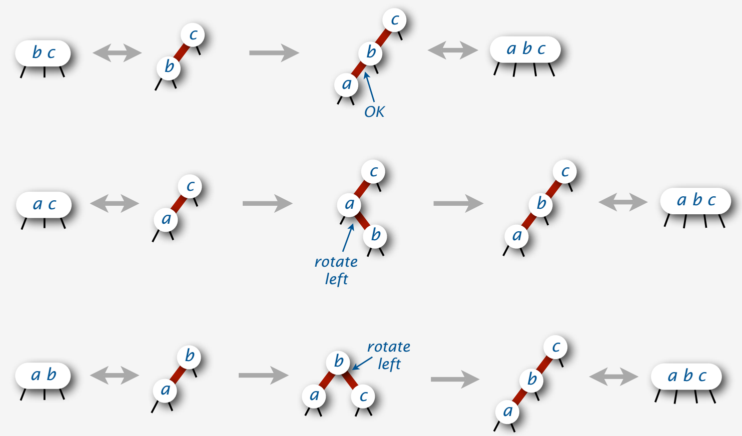
Case 1. Node at bottom is a 2-node.



Insertion in a LLRB tree: adding a node to the bottom

Invariant. Node at bottom is a 2-node or a 3-node.

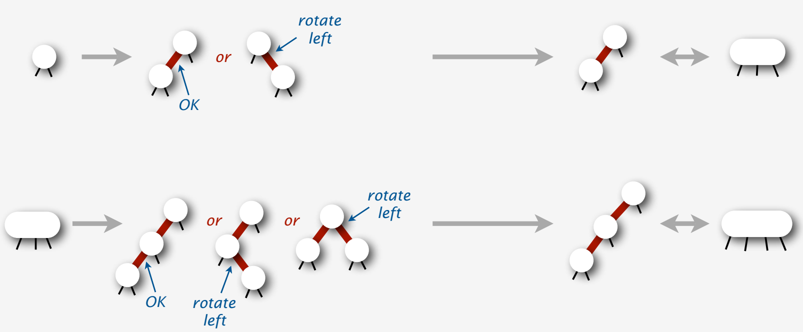
Case 2. Node at bottom is 3-node.



Insertion in a LLRB tree: adding a node to the bottom

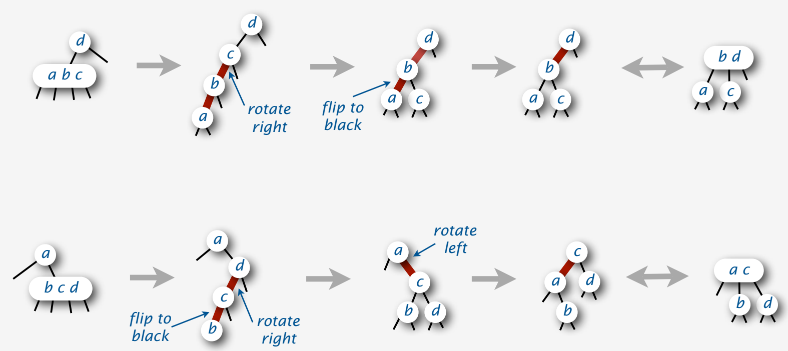
Key observation. Same code for all cases!

- Add new node at bottom as usual, with red link to glue it to node above.
- Rotate left if node leans right.



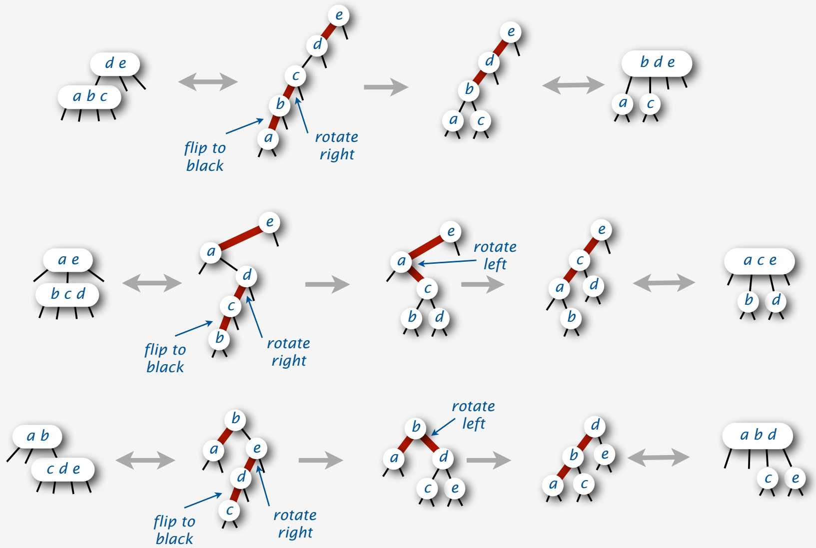
Insertion in a LLRB tree: splitting 4-nodes

Case 1. Parent of 4-node is a 2-node.



Insertion in a LLRB tree: splitting 4-nodes

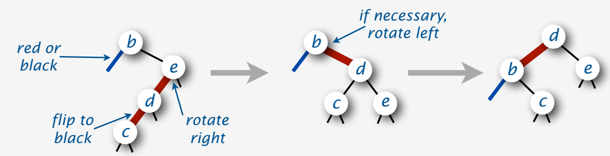
Case 2. Parent of 4-node is a 3-node.



Insertion in a LLRB tree: splitting 4-nodes

Key observation. Same code for all cases!

- Rotate right to balance the 4-node.
- Flip colors to pass red link up one level.
- Rotate left if necessary to make link lean left.

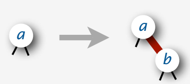


Insertion in a LLRB: strategy revisited

Basic strategy. Maintain 1-1 correspondence with 2-3-4 trees.

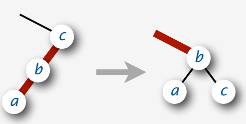
Search as usual.

- If key not found, insert a new node at the bottom.
- Might leave right-leaning link.



Split 4-nodes on the way down the tree.

- Right-rotate and flip color.
- Might leave right-leaning link.

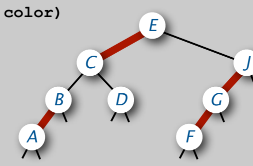


New trick: enforce left-leaning condition on the way up the tree.

- Left-rotate any right-leaning link on search path.
- Easy with recursion (do it after recursive calls).
- No other right-leaning links in tree.

Red-black tree implementation: Java skeleton

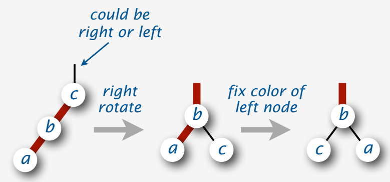
```
public class RedBlackBST <Key extends Comparable<Key>, Value>
{
   private static final boolean RED   = true;
   private static final boolean BLACK = false;
   private Node root;

   private class Node
   {
      private Node left, right;  // left and right subtrees
      private boolean color;     // color of parent link
      private Key key;           // key
      private Value val;         // value
      public Node(Key key, Value val, boolean color)
      {
         this.key   = key;
         this.val   = val;
         this.color = color;
      }
   }

   private boolean isRed(Node x)
   {
      if (x == null) return false;
      return (x.color == RED);
   }
}
```



Red-black tree implementation: basic operations

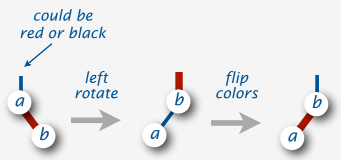
1. Split a 4-node.

```
private Node splitFourNode(Node h)
{
   x = rotR(h);
   x.left.color = BLACK;
   return x;
}
```



2. Enforce left-leaning condition.

```
private Node leanLeft(Node h)
{
   x = rotL(h);
   x.color = x.left.color;
   x.left.color = RED;
   return x;
}
```



Insertion in a LLRB tree: Java implementation

Remark. Only a few extra lines of code to standard BST insert.

```
private Node insert(Node h, Key key, Value val)
{
   if (h == null)
      return new Node(key, val, RED);

   if (isRed(h.left) && isRed(h.left.left))
      h = splitFourNode(h);

   int cmp = key.compareTo(h.key);
   if      (cmp  < 0) h.left  = insert(h.left,  key, val);
   else if (cmp  > 0) h.right = insert(h.right, key, val);
   else if (cmp == 0) h.val = val;

   if (isRed(h.right))
      h = leanLeft(h);

   return h;
}
```

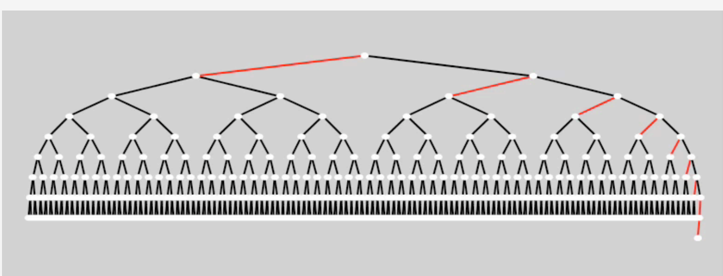
insert at bottom

split 4-nodes on the way down

standard insert

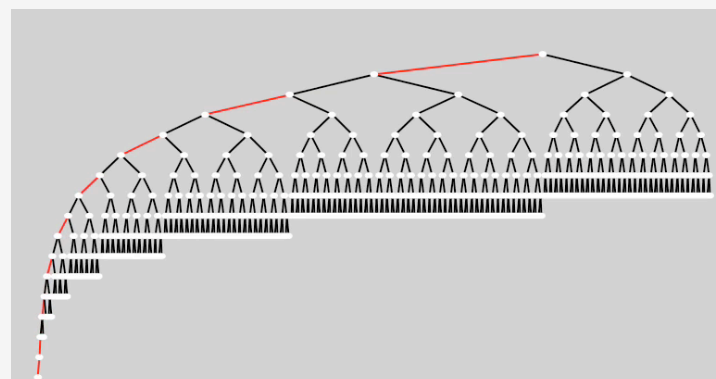
fix right-leaning reds on the way up

Insertion in a LLRB tree: visualization



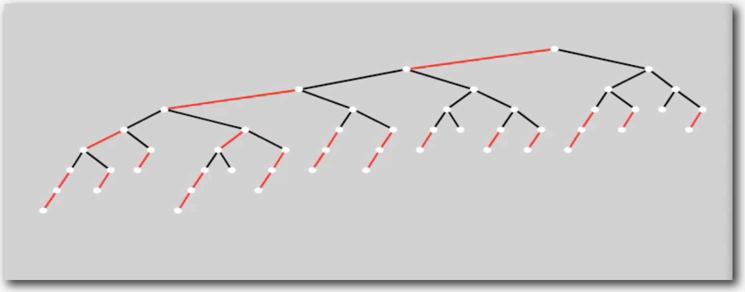
511 insertions in ascending order

Insertion in a LLRB tree: visualization



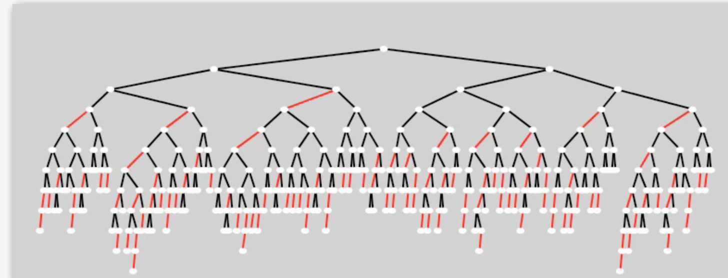
511 insertions in descending order

Insertion in a LLRB tree: visualization



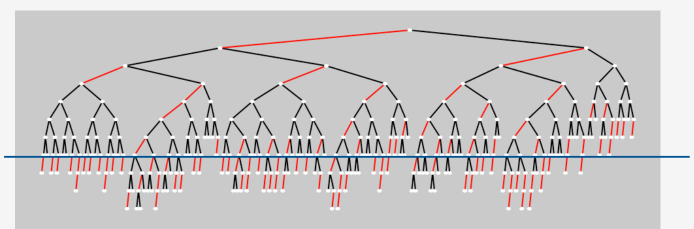
50 random insertions

Insertion in a LLRB tree: visualization

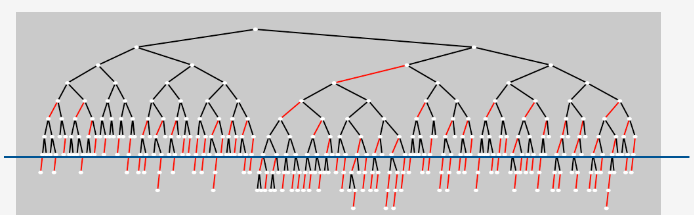


500 random insertions

Typical random LLRB trees



average node depth



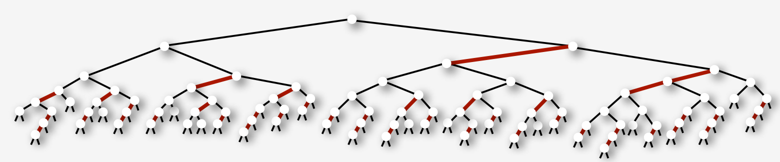
average node depth

Balance in left-leaning red-black trees

Proposition A. Every path from root to leaf has same number of black links.

Proposition B. Never three red links in-a-row.

Proposition C. Height of tree is less than 3 lg N + 2 in the worst case.



Property D. Height of tree is ~ lg N in typical applications.

Property E. Nearly all 4-nodes are on the bottom in the typical applications.

Why left-leaning trees?

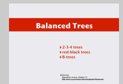
old code (that students had to learn in the past)

```
private Node insert(Node x, Key key, Value val, boolean sw)
{
   if (x == null)
      return new Node(key, value, RED);
   int cmp = key.compareTo(x.key);

   if (isRed(x.left) && isRed(x.right))
   {
      x.color = RED;
      x.left.color  = BLACK;
      x.right.color = BLACK;
   }
   if (cmp < 0)
   {
     x.left = insert(x.left, key, val, false);
     if (isRed(x) && isRed(x.left) && sw)
        x = rotR(x);
     if (isRed(x.left) && isRed(x.left.left))
      {
         x = rotR(x);
         x.color = BLACK; x.right.color = RED;
      }
   }
   else if (cmp > 0)
   {
      x.right = insert(x.right, key, val, true);
      if (isRed(h) && isRed(x.right) && !sw)
         x = rotL(x);
      if (isRed(h.right) && isRed(h.right.right))
      {
         x = rotL(x);
         x.color = BLACK; x.left.color = RED;
      }
   }
   else x.val = val;
   return x;
}
```

extremely tricky

new code (that you have to learn)

```
private Node insert(Node h, Key key, Value val)
{
   int cmp = key.compareTo(h.key);
   if (h == null)
      return new Node(key, val, RED);
   if (isRed(h.left) && isRed(h.left.left))
   {
      h = rotR(h);
      h.left.color  = BLACK;
   }
   if (cmp < 0)
      h.left = insert(h.left, key, val);
   else if (cmp > 0)
      h.right = insert(h.right, key, val);
   else
      x.val = val;
   if (isRed(h.right))
   {
      h = rotL(h);
      h.color = h.left.color;
      h.left.color = RED;
   }
   return h;
}
```



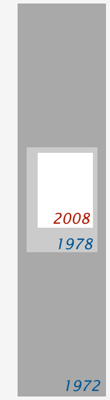
straightforward
(if you've paid attention)

Why left-leaning trees?

Simplified code.
- Left-leaning restriction reduces number of cases.
- Recursion gives two (easy) chances to fix each node.
- Short inner loop.

Same ideas simplify implementation of other operations.
- Delete min/max.
- Arbitrary delete.

Improves widely-used algorithms.
- AVL trees, 2-3 trees, 2-3-4 trees.
- Red-black trees.



Bottom line. Left-leaning red-black trees are the simplest to implement and fastest in practice.

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
unordered array	N	N	N	N/2	N	N/2	no	`equals()`
unordered list	N	N	N	N/2	N	N/2	no	`equals()`
ordered array	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
ordered list	N	N	N	N/2	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
randomized BST	3 lg N	3 lg N	3 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
2-3-4 tree	c lg N	c lg N	c lg N	c lg N	c lg N	c lg N	yes	`compareTo()`
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes	`compareTo()`

exact value of coefficient unknown
but extremely close to 1

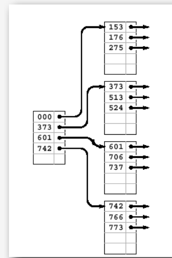
- 2-3-4 trees
- red-black trees
- B-trees

B-trees (Bayer-McCreight, 1972)

B-tree. Generalizes 2-3-4 trees by allowing up to M links per node.

Main application: file systems.

- Reading a page into memory from disk is expensive.
- Accessing info on a page in memory is free.
- Goal: minimize # page accesses.
- Node size M = page size.



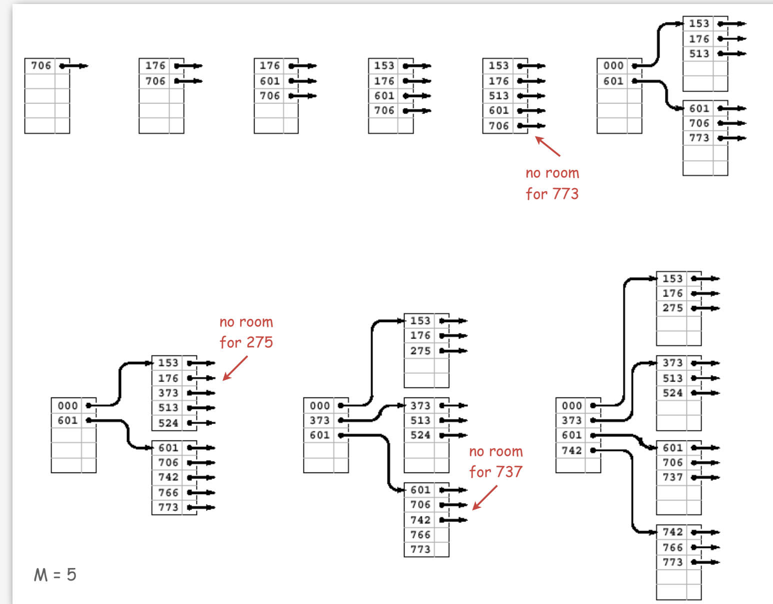
Space-time tradeoff.

- M large ⇒ only a few levels in tree.
- M small ⇒ less wasted space.
- Typical M = 1000, N < 1 trillion.

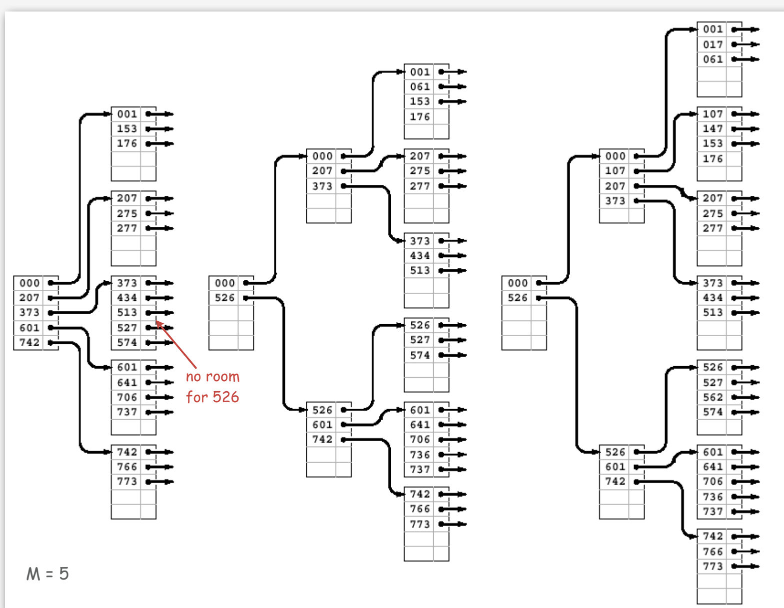
3 or 4 in practice (!)

Bottom line. Number of page accesses is $\log_M N$ per op in worst case.

B-Tree Example



B-Tree Example (cont)



Balanced trees in the wild

Red-black trees are widely used as system symbol tables.

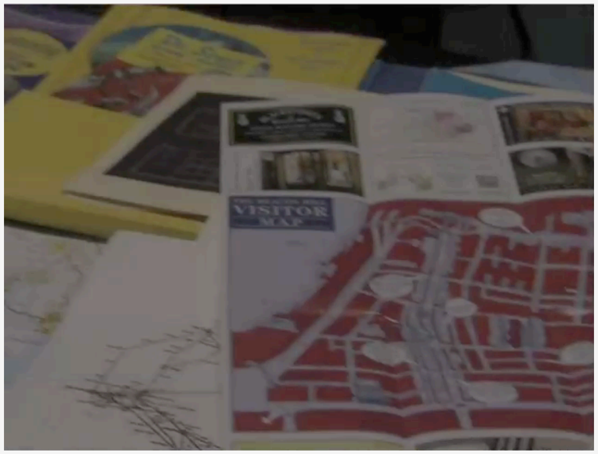
- Java: `java.util.TreeMap`, `java.util.TreeSet`.
- C++ STL: map, multimap, multiset.
- Linux kernel: completely fair scheduler, `linux/rbtree.h`.

B-tree variants. B+ tree, B*tree, B# tree, …

B-trees (and variants) are widely used for file systems and databases.

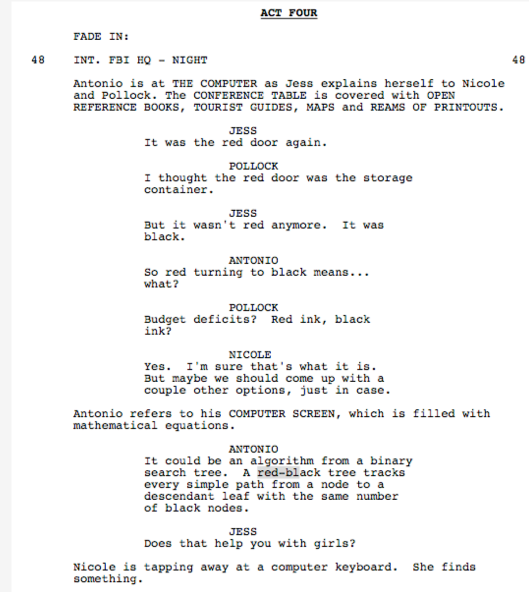
- Windows: HPFS.
- Mac: HFS, HFS+.
- Linux: ReiserFS, XFS, Ext3FS, JFS.
- Databases: ORACLE, DB2, INGRES, SQL, PostgreSQL.

Red-black trees in the wild



Common sense. Sixth sense.
Together they're the
FBI's newest team.

Red-black trees in the wild



ACT FOUR

FADE IN:

48 INT. FBI HQ - NIGHT 48

Antonio is at THE COMPUTER as Jess explains herself to Nicole and Pollock. The CONFERENCE TABLE is covered with OPEN REFERENCE BOOKS, TOURIST GUIDES, MAPS and REAMS OF PRINTOUTS.

JESS
It was the red door again.

POLLOCK
I thought the red door was the storage container.

JESS
But it wasn't red anymore. It was black.

ANTONIO
So red turning to black means... what?

POLLOCK
Budget deficits? Red ink, black ink?

NICOLE
Yes. I'm sure that's what it is. But maybe we should come up with a couple other options, just in case.

Antonio refers to his COMPUTER SCREEN, which is filled with mathematical equations.

ANTONIO
It could be an algorithm from a binary search tree. A red-black tree tracks every simple path from a node to a descendant leaf with the same number of black nodes.

JESS
Does that help you with girls?

Nicole is tapping away at a computer keyboard. She finds something.